Metaprograms to Help Expressions with their Problem

Seyed H. Haeri (Hossein)[1] and Paul Keir[2]

[1] Université catholique de Louvain, Belgium. email: `hossein.haeri@ucl.ac.be`
[2] University of the West of Scotland, UK. email: `paul.keir@uws.ac.uk`

Abstract. Integration of a decentralised pattern matching is a technique that enables a recent solution to the Expression Problem. The single former implementation of this technique was in Scala. In this paper, we highlight the C++ implementation of the same technique to solve the Expression Problem in C++. Unlike the former implementation which relies on stackability of Scala `trait`s, this new implementation relies on compile-time metaprogramming for automatic iterative pointer introspection at runtime. That iteration enables late binding using overload resolution, which the compiler is already capable of. The C++ implementation outperforms the Scala one by providing strong static type safety and offering considerably easier usage.

1 Introduction

The Expression Problem (EP) [13,41,51] is a recurrent problem in Programming Languages, for which a wide range of solutions have been proposed. Consider [49,29,46,31,5,52], to name a few. Haeri [18] phrases EP as the challenge of implementing an Algebraic Datatype (ADT) – defined by its cases and the functions on it – such that it:

E1. is *extensible in both dimensions*: Both new cases and functions can be added.
E2. provides *weak static type safety*: Applying a function f on a statically[3] constructed ADT term t should fail to compile when f does not cover all the cases in t.
E3. upon extension, forces *no manipulation or duplication* to the existing code.
E4. accommodates the extension with *separate compilation*: Compiling the extension imposes no requirement for repeating compilation or type checking of existing ADTs and functions on them. Compilation and type checking of the extension should not be deferred to the link or run time.

EP is recurrent because it is repeatedly faced over embedding DSLs – a task commonly taken in the PL community. Embedding a DSL is often practised in phases, each having its own ADT and functions defined on it. For example, take the base and extension to be the type checking and the object code generation

[3] If the guarantee was for dynamically constructed terms too, we would have called it strong static type safety.

phases, respectively. One wants to avoid recompiling, manipulating, and duplicating one's type checker if the object code generation adds more ADT cases or defines new functions on them.

It is common for EP solutions to take the following as the running example: An ADT NA with $\underline{N}$umbers and $\underline{A}$ddition as the only cases and another ADT NAM with $\underline{M}$ultiplication in addition to those of NA:

$$\begin{aligned}\alpha_{NA} &= \underline{N}um_{NA}(n) \quad \mid \underline{A}dd_{NA}(\alpha_{NA}, \alpha_{NA}),\\ \alpha_{NAM} &= \underline{N}um_{NAM}(n) \mid \underline{A}dd_{NAM}(\alpha_{NAM}, \alpha_{NAM}) \mid \underline{M}ul_{NAM}(\alpha_{NAM}, \alpha_{NAM}).\end{aligned}$$

Using $\gamma\Phi C_0$ [20] as the formalism, one denotes those as: $NA = \underline{N}um \oplus \underline{A}dd$ and $NAM = \underline{N}um \oplus \underline{A}dd \oplus \underline{M}ul$. Notice how the latter formalism dismisses the subscripts for Num, Add, and Mul. We use the $\gamma\Phi C_0$ notation hereafter.

A recent solution to EP (and a generalisation of it called the Expression Families Problem [30]) was given by Haeri and Schupp [21]. Their solution was based on a technique called integration of a decentralised pattern matching (IDPaM). Haeri and Schupp presented their solution in Scala. We deploy the same technique here to solve EP in C++.

Scala and C++ are different in many ways, causing significant differences between the Scala IDPaM and the C++ one. The Scala IDPaM uses type constraints and stackability of **`trait`**s. The C++ IDPaM uses **`template`** and macro metaprogramming. An overview of the C++ IDPaM at § 3.

Our contributions are:

- We implement the C++ IDPaM that offers easier roles[4] to take for solving EP than the Scala IDPaM (§ 7) and that better addresses the EP concerns (§ 4). Our key enabler for the relative ease of use is our one-liner macro for instructed late-binding (§ 5).
- We show that IDPaM can offer **strong** static type safety even in the absence of defaults and that stackability is not a cornerstone of IDPaM.
- We discuss how our technology is different from Generalised Algebraic Datatypes (GADTs) (§ 8).
- We provide the first manifestation of the roles to be taken for solving EP (§ 6). That insight enables us to systematically review related work § 9.

We achieve those because C++ metaprogramming allows us to encode the ADT in a way that: firstly, the ADT and its cases can be **programmatically** queried for one another; and, secondly, one can **programmatically** traverse the cases of an ADT for introspection.

The first point above makes definition of both ADTs and functions on them particularly less involved than the Scala IDPaM. That reduced involvement is significant. Recall that EP is the essence of challenges in embedding DSLs. During the latter exercise, that reduced involvement is a precious gift – especially, with automation of the embedding in mind.

We find the mere implementation of IDPaM in C++ constitutes valuable research. That is because, with the categorical differences between Scala and

[4] Cf. § 6 for the role definitions.

C++, this work demonstrates independence of IDPaM from Scala. Here are three differences that come to play in this work:

First, a cornerstone in Scala is its dependently typed methods – a feature not available in C++. Second, Scala offers no guarantee for any method call not to be late-bound. On the contrary, in C++, only calls to member functions through references or pointers are late-bound. Third, in the Scala IDPaM, the stackability of `trait`s takes a major role. The closest language feature that C++ offers to `trait`s is multiple inheritance, albeit with no stackability.

This paper assumes modest familiarity with C++17.

2 An Overview of the Scala IDPaM

It is tempting to use plain OOP for solving EP. The idea would be for ADT cases to inherit from the ADT itself and virtual functions to implement the functions defined on the ADT. That way, new cases could be defined by simple new derivations from the ADT. Furthermore, with virtual functions being readily late-bound by the compiler, runtime polymorphism would ensure the correct function behaviour on each ADT case. That approach fails to address E3 and E4 because addition of new functions amounts to recompiling the entire existing hierarchy and the depending code. We now review the Scala IDPaM [21]:

Case Components IDPaM too prescribes for the ADT cases to inherit from the ADT. But, IDPaM is also inspired by Component-Based Software Engineering (CBSE) [45, §17],[38, §10]. Like previous EP solutions of Haeri and Schupp [19,18,20], IDPaM takes a *components-for-cases* (C4C) approach: that is, each ADT case is implemented using a standalone component (in its CBSE sense) that is ADT-parameterised, a.k.a., *case components*. Here are the Scala case components for *Num* and *Add*:

```
trait IAE[E <: IAE[E]]
class Num[E <: IAE[E], N <: Num[E, N] with E](val n: Int)
class Add[E <: IAE[E], A <: Add[E, A] with E](val left: E, val right: E)
```

ADT Definition Armed with those, one defines *NA* as:

```
trait NA extends IAE[NA]
case class Num(n: Int) extends Num[NA, Num](n) with NA
case class Add(left: NA, right: NA) extends Add[NA, Add](left, right) with NA
```

Match Components Instead of virtual functions of OOP, for the implementation of functions defined on a datatype, IDPaM gets the programmer to instruct the late-binding. For example, here is how to obtain pretty-printing for `NA`:

```
object to_string extends PrB[NA] with PrN[NA] with PrA[NA] with PrF[NA]
```

`to_string` assembles building blocks provided by the programmer. (Cf. § 5 for the macro details.) In short, the assembled building blocks form a structure akin to the familiar pattern matching of functional programming. The familiar pattern matching, however, is holistic; all the match statements are together in the pattern matching and the individual match statements do not exist elsewhere. One can essentially not detach the individual match statements from the holistic pattern matching. Reusing the match statements is, hence, impossible. What we referred above to as the building blocks are, on the contrary, independent of the resulting assembly in which they set up. Those are, again, components in the CBSE sense. We call them the *match components*. `PrN` and `PrA` above are the pretty-printing match components of *Num* and *Add*:

```
class PrN[E <: IAE[E]] {
  override def the_to_string_match: E => String = {
    case (n: Num[_, _]) => n.toString
    case (e: E)         => super.the_to_string_match(e)
}}
class PrA[E <: IAE[E]] {
  override def the_to_string_match: E => String = {
    case (a: Add[_, _]) => to_string(a.left) + " + " + to_string(a.right)
    case (e: E)         => super.the_to_string_match(e)
}}
```

And, `PrB` and `PrF` are technical details required at the beginning and at the end of every integration that leads to definition of a function on an ADT.

```
class PrB[E <: IAE[E]] {
  def the_to_string_match: E => String = { throw ... }
  def to_string(e: E): String
}
class PrF[E <: IAE[E]] {
  override def to_string(e: E): String = super.the_to_string_match(e)
}
```

Each match component corresponds to one and only one match statement – enabling *decentralisation* of a pattern matching. The set of match statements and their order, in the Scala IDPaM, is open to the programmer for configuration at the right time. Instead of it being delivered holistically, the pattern matching then is by *integration* of the (decentralised) match components. Hence, the IDPaM name.

3 An Overview of the C++ IDPaM

Case Components In the C++ IDPaM, the ADT-parametrisation translates to type-parametrisation by ADT. For example, here are the case components for *Num* and *Add*:

```
template<typename ADT> struct Num: ADT    //Num α : ℤ → α
{ Num(int n): n_(n) {} int n_; };
template<typename ADT> struct Add: ADT {//Add α : α × α → α
  Add(const ADT& l, const ADT& r): l_(msfd<ADT>(l)), r_(msfd<ADT>(r)) {}
  const std::shared_ptr<ADT> l_, r_;
};
```

Notice how `Num` and `Add` both take `ADT` as a type parameter and inherit from it. This is a specific way of F-Bounding [9] commonly referred to in C++ as Mixins [50, §21.3]. (`msfd` is an implementation detail we drop the explanation of here due to space restrictions.[5] See § 5 for why the inheritance.)

[5] It performs similar iterations to those explained in § 5 to polymorphically assign a "pointer to the right case components" to "a pointer to the ADT."

ADT Definition The necessary steps for implementing *NA* in C++ IDPaM is:

```
1 template<> struct adt_cases<NA> { using type = std::tuple<Num<>, Add<>>; };
```

The above `template` specialisation of `adt_cases` for `NA` (line 1) is a meta-function instructing the compiler for `adt_cases` := `adt_cases` $\cup$ $\{$`NA` $\mapsto$ $Num \oplus Add\}$. That is, it introduces `std::tuple<Num<>, Add<>>` to the compiler as the *case list* of `NA`, enabling the compiler to infer the former from the latter, when required. Other case component combinations are done similarly, provided the presence of the additional case components.

Match Components Recall from § 2 that, for the implementation of functions defined on a datatype, IDPaM gets the programmer to instruct the late-binding. In the C++ IDPaM, this instruction is a one-liner that takes advantage of overload resolution – as readily available in C++. That instructed late-binding is a one-off for each function defined on ADTs, no matter how many new case components added later. For example, the one-off macro expansion for pretty-printing is

```
IMPL_DISPATCH_TO_MATCH_COMP(string, to_string)
```

Under the hood, the one-liner assembles match components. In the C++ IDPaM, the match components are simply function overloads. For example, the two overloads of `the_to_string_match` below are the two match components for the pretty-printing of `Num` and `Add` above:

```
template<typename ADT>                                   //to_string(Num(n)) = to_string(n)
string the_to_string_match(const Num<ADT>& n) {return std::to_string(n.n_);}
template<typename ADT>             //to_string(Add(l, r)) = to_string(l) + "+" + to_string(r)
string the_to_string_match(const Add<ADT>& a)
{return to_string(*a.l_) + " + " + to_string(*a.r_);}
```

Using an `operator""_n` and an `operator +` for syntactic sugaring, one gets "$5 + 5 + 4$" for `to_string(5_n + 5_n + 4_n)`, as expected.

4 Addressing The EP Concerns

Having an overview of the C++ IDPaM and the technology behind the one-liner, it is time to substantiate our claim about the C++ IDPaM solving the EP. We do so by going into the EP concerns one-by-one.

4.1 E1 (Bidimensional Extensibility)

Adding a new case is a matter of implementing a new case component. For example, a case for Multiplication can be provided by implementing a `Mul` just like `Add` in § 3:

```
template<typename ADT> struct Mul: ADT {
  Mul(const ADT& l, const ADT& r): l_(msfd<ADT>(l)), r_(msfd<ADT>(r)) {}
  /* ... See § 4.2 ... */ const std::shared_ptr<ADT> l_, r_;
};
```

A new ADT $NAM = \underline{N}um \oplus \underline{A}dd \oplus \underline{M}ul$ can then be implemented as easy as NA in § 3:

```
template<> struct adt_cases<NAM> {using type = std::tuple<Num<>, Add<>, Mul<>>;};
```

Note that `NAM` neither replaces nor shadows over `NA`. The above two ADT types are completely independent and can coexist even in the same **`namespace`**. It only is that, conceptually, `NAM` is a compatible extension [18] to `NA` because all the cases of the latter are also cases of the former. One can take that relationship between `NAM` and `NA` on board as:

```
template<> struct adt_cases<NAM>
{using type = tuple_type_cat<typename adt_cases<NA>::type, Mul<>>;};
```

where implementing `tuple_type_cat` for concatenation of types and `std::tuple` types is routine.

Pretty-printing for `Mul` simply adds the pertaining overload for `the_to_string_match` in § 3.

```
template<typename ADT> string the_to_string_match(const Mul<ADT>& m)
{return to_string(*m.l_) + " * " + to_string(*m.r_);}
```

Note that, upon addition of `Mul`'s match component, there is no need for expanding the one-liner again for `to_string`.

Finally, brand new functions like evaluation on the existing cases takes a new macro expansion to instruct late-binding (c.f., § 5) `IMPL_DISPATCH_TO_MATCH_COMP(int, eval)` and implementing the match components.

```
template<typename ADT> int the_eval_match(const Num<ADT>& n) {return n.n_;}
template<typename ADT> int the_eval_match(const Add<ADT>& a)
{return eval(*a.l_) + eval(*a.r_);}
template<typename ADT> int the_eval_match(const Mul<ADT>& m)
{return eval(*m.l_) * eval(*m.r_);}
```

4.2 E2 (Static Type Safety)

If the programmer forgets to include a case in the case list of an ADT, the compiler will not allow construction of terms using that case for that ADT.[6] Compilation of the expression `12_n * 14_n`, for example, will fail because of the following `static_assertion` in the body of `Mul`:

```
static_assert(is_case_in_adt<Mul<>, ADT>::value, ...);
```

Even attempting **`using`** `NAMul = Mul<NA>` will fail. Although, that is achieved using SFINAE[7] techniques that we do not present here due to space restrictions. If the respective match component of a case is not available (say, because it is forgotten), the code will be rejected at compile-time. For example, if the pretty-printing match component is not provided for `Mul`, the following compile-error will be produced for `NAM`: "no matching function for call to '`the_to_string_match(const Mul<NAM>&)`'."

[6] This applies to both statically **and** dynamically constructed terms; hence, strong static type-safety.

[7] Substitution Failure Is Not An Error [50, §8.4]

4.3 E3 (No Manipulation/Duplication)

Notice how nothing in the evidence for our support for E1 and E2 requires manipulation, duplication, or recompilation of the existing codebase. Our support for E3 follows.

4.4 E4 (Separate Compilation)

Our support for E4, in fact, follows just like E3. It turns out, however, that C++ **`template`**s enjoy two-phase translation [50, §14.3.1]: Their parts that depend on the type parameters are type checked (and compiled) only when they are instantiated, i.e., when concrete types are substituted for all their type parameters. As a result, type checking (and compilation) will be redone for every instantiation. That type-checking peculiarity might cause confusion w.r.t. our support for E4.

In order to dispel that confusion, we need to recall that `Add`, for instance, is a class **`template`** rather than a class. In other words, `Add` is not a type (because it is of kind $* \to *$) but `Add<NA>` is. The interesting implication here is that `Add<NA>` and `Add<NAM>` are in no way associated to one another. Consequently, introduction of `NAM` in presence of `NA`, causes no repetition in type checking (or compilation) of `Add<NA>`. (`Add<NAM>`, nonetheless, needs to be compiled in presence of `Add<NA>`.) The same argument holds for every other case component or match component already instantiated with the existing ADTs.

More generally, consider a base ADT $\Phi_b = \oplus\overline{\gamma}$ and its extension $\Phi_e = (\oplus\overline{\gamma}) \oplus (\oplus\overline{\gamma}')$. Let $\#(\overline{\gamma}) = n$ and $\#(\overline{\gamma}') = n'$, where $\#(.)$ is the number of components in the component combination. Suppose a C++ IDPaM codebase that contains case components for $\gamma_1, \ldots, \gamma_n$ and $\gamma'_1, \ldots, \gamma'_{n'}$. Defining Φ_b in such a codebase incurs compilation of n case components. Defining Φ_e on top incurs compilation of $n + n'$ case components. Nevertheless, that does not disqualify our EP solution because defining the latter component combination does not incur recompilation of the former component combination. Note that individual components differ from their combination. And, E4 requires the combinations not to be recompiled.

Here is an example in terms of DSL embedding. Suppose availability of a type checking phase in a C++ IDPaM codebase. Adding a type erasure phase to that codebase, does not incur recompilation of the type checking phase. Such an addition will, however, incur recompilation of the case components common between the two phases. Albeit, those case components will be recompiled for the type erasure phase. That addition leaves the compilation of the same case components for the type checking phase intact. Hence, our support for E4.

A different understanding from separate compilation is also possible, in which: an EP solution is expected to, upon extension, already be done with the type checking and compilation of the “core part” of the new ADT. Consider extending *NA* to *NAM*, for instance. With that understanding, *Num* and *Add* are considered the “core part” of *NAM*. As such, the argument is that the type checking and compilation of that “core part” should not be repeated upon the extension.

However, before instantiating `Num` and `Add` for `NAM`, both `Num<NAM>` and `Add<NAM>` are neither type checked nor compiled. That understanding, hence, refuses to take our work for an EP solution. We find that understanding wrong because the core of *NAM* is *NA*, i.e., the *Num* $\oplus$ *Add* **combination**, as opposed to both *Num* and *Add* but individually. Two quotations back our mindset up:

Zenger and Odersky [29] use the term "processors" for what we call "functions on datatypes." Their definition of separate compilation is as follows: "Compiling datatype extensions or adding new processors should not encompass re-type-checking the original datatype or existing processors." Observe how compiling `NAM` does not encompass repetition in the type checking and compilation of `NA`.

Wang and Oliveira [52] say an EP solution should support: "software evolution in both dimensions in a modular way, without modifying the code that has been written previously." Then, they add: "Safety checks or compilation steps must not be deferred until link or runtime." Notice how neither definition of new case components or ADTs, nor addition of case components to existing ADTs to obtain new ADTs, implies modification of the previously written code. Compilation or type checking of the extension is not deferred to link or runtime either. The situation is similar for the definition of new match components.

For more elaboration on the take of Wang and Oliveira on (bidimensional) modularity, one may ask: If *NA*'s client becomes a client of *NAM*, will the client's code remain intact under E3 and E4? Let us first disregard code that is exclusively written for *NA* for it is not meant for reuse by *NAM*:

```cpp
void na_client_f(const NA&) {...}
```

If on the contrary, the code only counts on the availability of *Num* and *Add*:

```cpp
template <
 typename ADT, typename = std::enable_if_t<adt_contains_v<ADT, Num<>, Add<>>>
> void na_plus_client_f(const ADT& x) {...}
```

Then, it can expectedly be reused upon transition from *NA* to *NAM*. (We drop the definition of `adt_contains_v` due to space restrictions.)

5 The One-Liner

Before the technical development of this section, we would like to explain a naming intention of ours: We chose the name "one-liner" because it takes only one line of code to **use** the macro `IMPL_DISPATCH_TO_MATCH_COMP` for instructing the late-binding. As we are about to see in this section, definition of the one-liner, however, takes multiple dozens of lines.

It now is time to reveal the technology behind our one-liner. We begin by presenting some utility macros used by the one-liner macro. The three macros below facilitate name production for the different role players in the one-liner technology. They all do so by juxtaposition of tokens during macro expansion.

```cpp
#define THE_1ST_2NDER(first, second) the_##first##second
```

Given a pair of tokens `first` and `second`, the above macro expands to `"the_"` followed by `first` followed by `second`. That will become handy in the next two macros.

```cpp
#define FUNC_MATCH(name) THE_1ST_2NDER(name, _match)
```

Given a token `name`, the above macro expands to `"the_"` followed by `name` followed by `"_match"`. This is due to a naming convention we follow: For a function `f` on ADTs, the match components need to be called `the_f_match`. For example, note with the second argument passed to the one-liner macro in § 3 being `to_string`, the match components are called `the_to_string_match`.

```
1  #define FUNC_DISPATCH(name) THE_1ST_2NDER(name, _dispatcher)
```

The above macro expands similarly to `FUNC_MATCH`. Again, this is due to a naming convention we use: For a function `f` on ADTs, an internal set of functions will be generated upon expansion of the one-liner macro that are all called `the_f_dispatcher`. Those functions are at the core of the automation provided by the one-liner. We now present them along with the one-liner macro itself:

```
1  #define IMPL_DISPATCH_TO_MATCH_COMP(return_type, function_name)                \
```

The one-liner macro takes two parameters: the name of the function on ADTs (`function_name`) and the return type of the function (`return_type`).

```
2  template<typename, typename> struct FUNC_DISPATCH(function_name);              \
```

The one-liner builds on a class **`template`** that we call the *dispatcher*. The above two lines declare the right-named dispatcher for the compiler. The exact name of the dispatcher is determined using `FUNC_DISPATCH(function_name)`, which we explained earlier. The dispatcher takes two type parameters (line 2). The first one is the ADT for which dispatching is to take place. The second one represents the cases of that ADT that are thus far not examined. (More on what we mean by examination of the ADT cases right below.) Two scenarios are possible. Hence, the two **`template`** specialisations below:

```
3  template<typename ADT, typename C, typename... Cs>                             \
4  struct FUNC_DISPATCH(function_name)<ADT, std::tuple<C, Cs...>> {               \
5    static return_type match(const ADT& x) {                                     \
6      const auto cp = dynamic_cast<const C*>(&x);                                 \
7      if(cp) return FUNC_MATCH(function_name)(dynamic_cast<const C&>(*cp));       \
8      else return FUNC_DISPATCH(function_name)<ADT, std::tuple<Cs...>>::match(x);\
9    }                                                                            \
10 };                                                                             \
```

The first scenario is when there is at least one case of the `ADT` that is still not examined. The above code manifests that scenario in line 4 by taking the second type parameter of the dispatcher to be `std::tuple<C, Cs...>>`. As such, it names the first such `ADT` case `C`.

The duty of the `match` member function above is to determine whether the provided reference to the `ADT` is a reference to `C`. Only when the **`dynamic_cast`** at line 6 returns a valid pointer, the answer is positive. In that case, the reference to the `ADT` is safely cast to a reference to `C`. That cast is the critical point where the built-in overload resolution kicks in.

Roughly put, at line 6, the compiler knows that `x` is an instance of the case `C` (of the `ADT`). The built-in overload resolution, thus, can choose the correct overload out of the provided match components upon the call in line 7.

When the **`dynamic_cast`** fails, the dispatcher moves on to the next cases, if any (line 8). That is done by calling itself recursively on `x`, albeit with the remaining `ADT` cases (`std::tuple<Cs...>`) as the second type parameter.

```
11 template<typename ADT> struct FUNC_DISPATCH(function_name)<ADT, std::tuple<>> {\
12   static return_type match(const ADT& x)                                       \
13   { throw std::invalid_argument("unhandled case"); }                            \
14 };                                                                              \
```

The second scenario is when the dispatcher runs out of the ADT cases. That is manifested above by the dispatcher taking an empty case list (std::tuple <>) as the second type parameter (line 11). As will become clear below, that constitutes an internal error in our codebase, hence, the exception in line 13.

```
15 template<typename ADT, typename = std::enable_if_t<is_adt<ADT>::value>>        \
16 return_type function_name(const ADT& x)                                         \
17 {return FUNC_DISPATCH(function_name)<ADT,rendered_adt_cases_t<ADT>>::match(x);}\
```

We now need to recall from § 3 that the call to pretty-printing takes place via a function called to_string, rather than the_to_string_match or the_to_string_dispa-tcher. The snippet above generates that to_string. More generally, suppose that a function f on ADTs is passed as the second argument to the one-liner function. Then, the above snippet generates f by constructing a *call centre* that relays the task to the dispatcher.

The call centre has two overloads. The above overload is the one that is chosen when all that is **statically** known about x is that it is an instance of an ADT (line 15). In that situation, the dispatcher is called on x with two type arguments: the ADT (line 17) as well as its case list (line 17). (No explanation here on rendered_adt_cases_t due to space restrictions.)

The reason why the above overload of the call centre is required might not be immediate to the reader. To obviate that need, we revisit the code from § 3:

```
1 template<typename ADT> struct Add: ADT {
2   Add(const ADT& l, const ADT& r): l_(msfd<ADT>(l)), r_(msfd<ADT>(r)) {}
3   const std::shared_ptr<ADT> l_, r_;
4 };
5 template<typename ADT> string the_to_string_match(const Add<ADT>& a)
6 {return to_string(*a.l_) + " + " + to_string(*a.r_);}
```

Notice how the data members l_ and r_ (line 2) of Add are both shared pointers to ADT. The implication is that the static type available to the compiler for *a.l_ and *a.r_ (line 6) is ADT. As such, it is the above overload of the call centre that is called for the recursion at line 6.

```
18 template                                                                        \
19 <                                                                               \
20  typename C, typename = C, typename = std::enable_if_t<is_case<C>::value>      \
21 > return_type function_name(const C& x)                                         \
22 { return function_name<typename C::adt_type>(x); }
```

Finally, the programmer may also invoke the call centre with an argument, the exact ADT case of which is statically known by the compiler. As an example, take to_string(5_n + 5_n + 4_n) seen in § 3.

The latter overload of the call centre is chosen in that scenario because of enable_if_t<is_case<C>::value> in line 20. This overload of the call centre simply relays the job to the former overload by making the compiler take x as an instance of its ADT type. The ADT type of x is calculated using **typename** C:: adt_type in line 22. Note that upcasting x to its ADT type is valid because every case component inherits from its ADT. (Definitions of `is_case` and `adt_type` are dropped due to space restrictions.)

We are now in good position to define what we called "instructed late-binding" thus far. Reconsidering the first scenario above, one notices the similarity between that logic and that of the ordinary late-binding. Notwithstanding, the ordinary late-binding is an automatic service shipped by an object-oriented language's compiler, which the programmer takes for granted. On the contrary, by using the one-liner as a library facility, the programmer gets the first scenario to instruct the late-binding.

6 Roles

When it comes to comparison between different EP solutions, it is natural to choose the proficiency in addressing E1–E4 as the basis. Proficiency, here, is often understood as the relative effort required for addressing those EP concerns. We argue that it is meaningful to also compare EP solutions based on the facilities they offer to each role.

To that end, one needs to first study the roles that are to be taken for an EP solution to develop. The following roles come to mind:

– EP Solver: Brings a particular discipline to the implementation of ADTs and functions defined on them that solves EP.
– Solution User: Implements the ADTs and functions defined on them by submitting to the discipline brought by the EP solver.
– ADT User: Employs the solution user's ADTs and functions on them to create ADT terms and apply functions on those terms.

EP solutions often do not distinguish between the above roles.

Whilst exercising the IDPaM discipline, the solution user takes the case components and the match components off-the-shelf. As such, one can imagine two more roles in IDPaM: the case component vendor and the match component vendor. The IDPaM solution user also requires a medium for integrating the match components. In the C++ IDPaM, that medium is the one-liner detailed in § 5, which is provided as a one-off library facility.

7 Comparison with the Scala IDPaM

We now compare the Scala and the C++ IDPaM based on the roles in § 6.

EP Solver In the Scala IDPaM, ADT cases need to both inherit from the respective case component and the ADT. For example, the *Num* of *NA* in § 2 inherits both from `Num` (the case component) and `NA`. That is the **`case class`** `Num` in line 2 above. Additionally, due to technicalities of F-Bounding in Scala, the **`case class`** `Num` requires to tie the recursive type knot by substituting itself for the (second) type parameter of the `Num` case component.

In the Scala IDPaM, the programmer instructs the integration by **manual** assembly of match components in a feature-oriented fashion (cf. `to_string`, § 2).

With components having no definite representative in current programming languages, one needs to leverage other residents of the language to simulate components. In comparison to the Scala IDPaM, in the C++ IDPaM, the discipline required for component development is slightly more lightweight because it requires no type constraints. With C++ being a more verbose language, the Scala implementation is, however, more succinct.

The Scala IDPaM utilises **`trait`**s for match components. It uses **`trait`** stackability and **`super`** calls to enable integration. When a match component of theirs cannot handle the given task, it performs a **`super`** call, hoping that an upper match component in the stack will pick the task up.

The Scala IDPaM addresses all the EP concerns except E2. It is only in the presence of defaults [54] that it can also address that concern. That is, upon integration, if the programmer forgets to mix-in the respective match component of a case, only a runtime exception will be thrown. This is not totally unacceptable in object-oriented EP solutions, e.g., Lightweight Modular Staging (LMS) [42], MVCs [30], and Torgersen's second solution [49].

On the contrary, in the C++ IDPaM, E2 and the other EP concerns are all addressed. This is an important milestone because it proves that lack of support for E2 is not central to IDPaM. More to the point is IDPaM's capability of providing strong static type safety, even in the absence of defaults.

In comparison to the Scala IDPaM, understanding how the C++ IDPaM comes to be an EP solution is admittedly more difficult. This is because the Scala IDPaM only uses **`trait`**s: a lay feature of Scala. Our usage of **`template`** and macro metaprogramming, on the contrary, is relatively advanced in C++. That makes the C++ IDPaM less accessible in comparison to the Scala one.

Solution User Implementing ADTs in the Scala IDPaM is considerably more involved than the C++ IDPaM. The order of invoking match components, in the Scala IDPaM, gets fixed upon the integration. In the C++ IDPaM, on the contrary, the order is undetermined to the programmer and the compiler chooses to invoke match components according to the component combination in the term.

A distinctive difference between the C++ and the Scala IDPaM is as follows: In the latter work, for **every** ADT, the programmer is required to assemble the match components – even when the new ADT has the same case list of an existing ADT. For example, for an ADT $NA_2 = \mathit{Num} \oplus \mathit{Add}$, trying the object `to_string` above to terms of NA_2 will fail to compile.

On the contrary, the one-liner macro of the C++ IDPaM is only required to be expanded once for a given function to work for all ADTs ever. Besides, using the case list of a given ADT, the C++ compiler **automatically** assembles the right match components. This facility is available because, in C++, the programmer can define metafunctions too, i.e., functions on types that operate at the compile time exclusively.

ADT User Both Scala and C++ facilitate syntax sugaring to a great deal so that using ADTs takes comparable efforts across the two languages.

Component Vendors Match components are simple overloads of **template** functions in the C++ IDPaM. In the Scala IDPaM, on the other hand, match components are type parameterised **trait**s with a method named according to a naming convention that uses Scala's built-in pattern matching and **super** calls. Relying on that built-in support buys extra simplicity for the Scala IDPaM when the match components require other type parametrisation than the ADT they are integrating over. In the C++ IDPaM, we need special treatment for dealing with that. Providing case components, on the other hand, takes much less given the syntactic noise of C++.

8 C4C versus GADTs

When embedding DSLs, it is often convenient to piggyback on the host language's type system. In such practice, GADTs are a powerful means to guarantee the absence of certain type errors. For example, here is an excerpt from a Scala transliteration[8] of Kennedy and Russo's running example for GADTs in object-oriented languages [23]:

```
sealed abstract class Exp[T]
case class Lit(i: Int)                                       extends Exp[Int]
case class Plus(e1: Exp[Int], e2: Exp[Int])                  extends Exp[Int]
case class Equals(e1: Exp[Int], e2: Exp[Int])            extends Exp[Boolean]
case class Cond(e1: Exp[Boolean], e2: Exp[Int], e3: Exp[Int]) extends Exp[Int]
/* ... more case classes ... */  def eval[T](exp: Exp[T]): T = exp match {...}
```

Notice first that `Exp` is type parameterised, where `T` is an arbitrary Scala type. That is how `Lit` can derive from `Exp[Int]` whilst `Equals` derives from `Exp[Boolean]`. Second, note that `Plus` takes two instances of `Exp[Int]`. Contrast that with the familiar encodings of $\alpha = \mathit{Plus}(\alpha, \alpha) \mid \ldots$, for some ADT α. Unlike the GADT one, the latter encoding cannot outlaw nonsensical expressions such as `Plus(Lit(5), Lit(true))`. Third, note that `eval` is polymorphic in the carrier type of `Exp`, i.e., `T`.

The similarity between the above case definitions and our case components is that they are both type parameterised. Nevertheless, the former are parameterised by the type of the Scala expression they carry. Whereas, our case components are parameterised by their ADT types. The impact is profound. Suppose, for example, availability of a case component `Bool`with the corresponding **operator** `""_b`. In their current presentation, our case components cannot statically outlaw type-erroneous expressions like `12_n + "true"_b`. On the other hand, the GADT `Cond` is exclusively available as an ADT case of `Exp` and cannot be used for other ADTs.

The bottom line is that GADTs and C4C encodings of ADTs are orthogonal. One can always generalise our case components so they too are parameterised by their carrier types and so they can guarantee similar type safety. We have already implemented that idea. Future publications will report the result.

[8] Posted online by James Iry on Wed, 22/10/2008.

9 Related Work

In this section, we study three groups of related work: EP solutions are in § 9.1. Usage of similar techniques for solving other problems is in § 9.2. And, finally, the language definitional frameworks that offer facilities that can be used to solve EP in a C4C fashion are in § 9.3. The literature is sizeable for the first and third topic. We only selectively review those we deem to be in close proximity.

9.1 Expression Problem

Object Algebras Using object algebras [17] to solve EP has become popular over recent years. Oliveira and Cook [31] pioneered that. Oliveira et al. [34] address some awkwardness issues faced upon composition of object algebras. Rendel, Brachthäuser and Ostermann [40] add ideas from attribute grammars to get reusable tree traversals. An often neglected factor about solutions to EP is the complexity of term creation. That complexity increases from one work to the next in the above literature. The symptom develops to the extent that it takes Rendel, Brachthäuser and Ostermann 12 non-trivial lines of code to create a term representing "$3 + 5$". Of course, those 12 lines are not for the latter task exclusively and enable far more reuse. Yet, with the particular goal in their §5.3, those 12 lines are inevitable for term creation for "$3 + 5$", making that so heavyweight. Other recent works on object algebras [55,56] also suffer from that symptom. All in all, the ADT user has a considerably more involved job using the current object algebras technologies for EP. The solution user is a slightly heavier role with current object algebras for EP. For example, pretty-printing takes 12 (concise) Scala lines in [40], whereas that is 6 (syntactically noisy) C++ lines in ours. The EP solver's effort, however, is comparable to our solution.

Type Classes Swierstra's Datatypes à la Carte [46] uses Haskell's type classes to solve EP. In his solution too, ADT cases are ADT-independent but ADT-parameterised. He uses Haskell Functors to that end. Defining functions on ADTs amounts to defining a type class, instances of which materialising match statements for their corresponding ADT cases. Without syntactic sugaring, term creation become much more involved than that for ordinary ADTs. Defining the syntactic sugar takes many more steps than us, but, makes term creation straightforward. Using the Scala type classes [32] can lead to simpler syntactic sugar definition but needs extra work for the lack of direct support in Scala for type classes. As for the EP Solver, because type classes are widely-understood in the programming languages community, Swierstra's discipline is preferable with that background. Ours might be preferable for a mainstream language programmer. For the Solution User, the effort required is comparable across Swierstra's work and that of ours. The ADT User is likely to choose the C++ IDPaM over Swierstra for the above syntactic sugaring overhead.

Bahr and Hvitved extend Swierstra's work by offering Compositional Datatypes (CDTs) [4]. They aim at higher modularity and reusability. CDTs support more recursion schemes, and, extend to mutually recursive data types and GADTs.

Besides, syntactic sugaring is much easier using CDTs because smart constructors can be automatically deduced for terms. CDTs rely on Haskell features that are less widely-understood. EP Solver using CDTs is, therefore, a more complicated role in comparison to that of Swierstra or ours. Solution User is a similar role using CDTs to that using Swierstra's technology. ADT User is comparable of a role with the C++ IDPaM. Later on, Bahr and Hvitved offer Parametric CDTs (PCDTs) [5] for automatic α-equivalence and capture-avoiding variable bindings. PCDTs achieve that using Difuntors [27] (instead of Functors) and a CDT encoding of Parametric Higher-Order Abstract Syntax [10]. Case definitions take two phases: First an equivalent of our case components need to be defined. Then, their case components need to be materialised for each ADT, similar to but different from that of Haeri and Schupp [19,18]. With their clever Haskell usage and wide list advanced concepts, PCDTs have limited accessibility for the EP Solver. In order to take the benefits of PCDTs over CDTs into action, the Solution User has to put extra effort in, when compared with CDTs. Taking ADT User's role is similar to CDTs.

Modular Reifiable Matching (MRM) [33] improves Swierstra's work by targeting the type class' lack of desirable subtyping between core ADTs and functions defined on them and those of extensions. This is done via a new representation for two-level types [43] based on a list-of-functors. MRM also accommodates inheritance and overriding of functions defined on ADTs. That is achieved by storing match statements individually and combining them using generalised f-algebras. MRM's use of generic smart constructors makes definition of suitable syntax sugaring trivial.

The distinctive difference between C4C and the works of this group of related work is the former's inspiration by CBSE. Components, in their CBSE sense, ship with their 'requires' and 'provides' interfaces. Whereas, even though the latter works too parameterise cases by ADTs, the interface that CDTs, for instance, define do not go beyond algebraic signatures. Although we do not present that here, C4C can go well beyond that, enabling easy solutions to the Expression Families Problem [31] and Expression Compatibility Problem [21].

Other Garrigue [16] solves EP using global case definitions that, at their point of definition, become available to every ADT defined afterwards. Per se, a function that pattern matches on a group of these global cases can serve any ADT containing the selected group. OCaml's built-in support for Polymorphic Variants [16] makes all the EP roles easier to take. However, we minimise the drawbacks [6] of ADT cases being global by promoting them to components. That drawback aside, taking all the EP roles is easier in Garrigue's solution than that of ours.

Rompf and Odersky [42] employ a fruitful combination of the Scala features to present a very simple yet effective solution to EP using LMS. The support of LMS for E2 can be broken using an incomplete pattern matching. Yet, given that pattern matching is dynamic, whether LMS really relaxes E2 is debatable. Ease of taking the EP roles is comparable between LMS and us.

Although not quite related in the techniques used for solving EP, the recent work of Wang and Oliveira [52] is worth noting as well. This latter work is outstanding in its ease of use and the little number of advanced features it expects from the host language. Their approach uses covariant type refinement of return types. We, on the other hand, use (compile-time) metaprogramming with C++ templates and macros. Their solution is generally easier in all the roles than the C++ IDPaM.

9.2 Technique

Iterating a list of types for pointer introspection is not new. Alexandrescu [3, §11] used that technique first for a dozen of ways to enable multiple dispatch in C++. His use of the technique, nevertheless, does not take advantage of overload resolution and builds upon callback functions explicitly provided by the programmer for each dispatch case. Such similarities in the technique also exist between the works on Open Multi-Methods [36,37,44] and that of ours.

Finally, Wonnacott [53] too has similar concerns to E1 for enabling multiple dispatch. Yet, his work is not an EP solution in that it proceeds by recommending a change to the common virtual table technologies of the time.

9.3 Language Definitional Frameworks

Maude [11] serves as a platform for language development using Equational and Rewriting Logic [28,8]. The *parameterised modules* of Full Maude [12, Part II] can serve EP in a fashion akin to C4C. This is because they can be parameterised over *theories* – a concept similar to ordinary interfaces of OOP. Moreover, parameterised modules can be combined using multiple inheritance, serving additive composition [45, §17.3] in the fashion of the Scala IDPaM. In essence, Maude's parameterised modules are like built-in compositional object algebra interfaces [34] or verified software product-lines [47,48]. The inheritance can also serve as a sequential composition [45, §17.3] means for reuse of ADT cases, like the Scala LMS. If one spares only one ADT case per module, theory parametrisation can be leveraged as the ADT-parametrisation of C4C. Likewise, one can obtain match components by reducing the number of functions per module to one. Maude's *mix-fix* operators provide outstanding flexibility for term creation and thus the ADT user.

Similar to Maude, AspectLISA [39] can be employed for a C4C solution to the EP. Here is how: AspectLISA supports multiple attribute grammar inheritance [35]. (That is multiple inheritance for attribute grammars.) And, the `advice` facility of AspectLISA – which is used for traversal of terms built using attribute grammars – can be parameterised over both terminals and non-terminals of the grammar. The former can be used like how the C++ LMS combines case components to define an ADT. The latter can serve as a means for ADT parametrisation like case components.

MontiCore [24] uses the familiar OOP inheritance for ADT extension. The derived ADT – i.e., the extension – overrides a specific method of the parent ADT – i.e., the core – in which new match statements can be specified. The body of such a method typically consists of a series of visitors, each (of which) predesignated for a specific match. In the terminology of MontiCore, this series of visitors is said to produce a "compositional visitor." A visitor in MontiCore is like the match components of IDPaM. Hence, although not designed for this particular purpose, with the compositional visitors, MontiCore may look like built-in support for C4C. However, MontiCore fails to address E2 because its visitor composition is based on Java reflections, with no static safety.

DeepFJig [14] integrates modular composition and nesting into Featherweight Java [22]. This is achieved by offering a full stack of composition operators, inspired by Bracha's Jigsaw [7]. As a formalism, DeepFJig is very similar to but more powerful than $\gamma\Phi C_0$ (i.e., the formalism behind C4C solutions). In particular, DeepFJig's "`[+]`" generalises $\gamma\Phi C_0$'s "$\oplus$" in scope. In their DeepFJig presentation, the authors present a handful of EP solutions to demonstrate expressiveness of their language. By coincidence, their very last solution is of particular proximity to the C++ IDPaM, albeit using the built-in DeepFJig composition operators. These operators are powerful enough to dismiss the need for ADT parametrisation. But, the implication is downgrading of case components to nested classes. Obviously, because of their dependency to the enclosing class, nested classes are less independent than components.

GLoo [26] has an open-ended meta-level language that can be used for ADT specification. Furthermore, Gloo offers *mini-parsers* which can be used for implementing match components. The GLoo meta-level language is a dynamic functional language with constructs that are generally similar to the familiar OOP constructs. For further general purpose programming tasks, GLoo also facilitates interaction with Java. GLoo's *form extension operator* [25] can be used for ADT composition. The excess of syntactic noise in GLoo's meta-level language makes detailed assessment of its EP support very difficult – especially, with its lack of sufficient documentation.

Finally, MetaMod [15] has a built-in support for EP, with particular focus on DSL workbenching. That is achieved by making ADTs extensible by-design and allowing MetaMod processing units (i.e., functions defined on an ADT) to be declared *related* to both a group and an aspect of processing units. That grants functions on ADTs access to the internals of other functions, implying the desirable reuse.

10 Conclusion

We solve EP in C++ using macro and template metaprogramming for implementing IDPaM. Our solution outperforms its Scala predecessor in that it also statically rejects non-exhaustive function application on an ADT term (constructed statically **or** dynamically), even in the absence of defaults. Furthermore, it is way easier to employ. The cornerstone of the above outperformance is the instructed late-binding. C++ metaprogramming makes that possible by allowing: (1) programmatic queries over an ADT and its case list for one another; and, (2) programmatic traversal of the latter for introspection. Our solution has been tested for a set of 12 core lazy calculi. Due to space restrictions, however, we cannot provide details here.

It turns out that the same technique gives very clean solutions to various generalisations of EP as well as multiple dispatch. Moreover, we believe the use of components in our fashion can make the job of the ADT user considerably easier for object algebras. That is subject for future research. The material in this paper has thus far only been used for toy examples. It is our aim to benchmark it for embedding medium- and large-scale DSLs.

References

1. Multi-Methods for a C++ Encoding of ADTs. Under Review for the 16th APLAS, June 2018.
2. Solving the Expression Problem in C++, à la LMS. Under Review for the 15th FACS, October 2018.
3. A. Alexandrescu. *Modern C++ Design: Generic Prog. & Design Patterns Applied.* AW Longman Pub., Boston, MA, USA, 2001.
4. P. Bahr and T. Hvitved. Compositional Data Types. In J. Järvi and S.-C. Mu, editors, 7th *WGP*, pages 83–94, Tokyo, Japan, September 2011. ACM.
5. P. Bahr and T. Hvitved. Parametric Compositional Data Types. In J. Chapman and P. B. Levy, editors, 4th *MSFP*, volume 76 of *ENTCS*, pages 3–24, February 2012.
6. A. P. Black. The Expression Problem, Gracefully. In M. Sakkinen, editor, *MASPEGHI@ECOOP 2015*, pages 1–7. ACM, July 2015.
7. G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, Dept. CS, U. of Utah, 1992.
8. R. Bruni and J. Meseguer. Semantic Foundations for Generalized Rewrite Theories. *TCS*, 360(1-3):386–414, 2006.
9. P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In 4th *FPCA*, pages 273–280, September 1989.
10. A. Chlipala. Parametric Higher-Order Abstract Syntax for Mechanized Semantics. In J. Hook and P. Thiemann, editors, 13th *ICFP*, pages 143–156, Victoria, BC, Canada, September 2008.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, editor, 14th *RTA*, number 2706 in LNCS, pages 76–87. Springer-Verlag, June 2003.
12. M. Clavel, F. Durán, S. Eker, and J. Meseguer. *All about Maude: A High-Performance Logical Framework.* Springer-Verlag Berlin Heidelberg, February 2007.
13. W. R. Cook. Object-Oriented Programming Versus Abstract Data Types. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *FOOL*, volume 489 of *LNCS*, pages 151–178, Noordwijkerhout (Holland), June 1990.
14. A. Corradi, M. Servetto, and E. Zucca. DeepFJig: Modular Composition of Nested Classes. *JOT*, 11(2):1: 1–42, 2012.
15. A. M. Şutîi, M. van den Brand, and T. Verhoeff. Exploration of Modularity and Reusability of Domain-Specific Languages: An Expression DSL in MetaMod. *CLSS*, 51:48–70, 2018.

16. J. Garrigue. Code Reuse through Polymorphic Variants. In *FSE*, number 25, pages 93–100, 2000.
17. Guttag, J. V. and Horning, J. J. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.
18. S. H. Haeri. *Component-Based Mechanisation of Programming Languages in Embedded Settings*. PhD thesis, STS, TUHH, Germany, December 2014.
19. S. H. Haeri and S. Schupp. Reusable Components for Lightweight Mechanisation of Programming Languages. In W. Binder, E. Bodden, and W. Löwe, editors, 12th *SC*, volume 8088 of *LNCS*, pages 1–16. Springer, June 2013.
20. S. H. Haeri and S. Schupp. Expression Compatibility Problem. In J. H. Davenport and F. Ghourabi, editors, 7th *SCSS*, volume 39 of *EPiC Comp.*, pages 55–67. EasyChair, March 2016.
21. S. H. Haeri and S. Schupp. Integration of a Decentralised Pattern Matching: Venue for a New Paradigm Intermarriage. In M. Mosbah and M. Rusinowitch, editors, 8th *SCSS*, volume 45 of *EPiC Comp.*, pages 16–28. EasyChair, April 2017.
22. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.
23. A. Kennedy and C. V. Russo. Generalized Algebraic Data Types and Object-Oriented Programming. In R. E. Johnson and R. P. Gabriel, editors, 20th *OOPSLA*, pages 21–40, San Diego, CA, USA, October 2005. ACM.
24. H. Krahn, B. Rumpe, and S. Völkel. MontiCore: a Framework for Compositional Development of Domain Specific Languages. *STTT*, 12(5):353–372, 2010.
25. M. Lumpe. A Lambda Calculus with Forms. In T. Gschwind, U. Aßmann, and O. Nierstrasz, editors, 1st *SC*, volume 3628 of *LNCS*, pages 83–98. Springer, April 2005.
26. M. Lumpe. Growing a Language: The GLoo Perspective. In C. Pautasso and É. Tanter, editors, 7th *SC*, volume 4954 of *LNCS*, pages 1–19. Springer, 2008.
27. E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. In J. Williams, editor, 7th *FPCA*, pages 324–333, La Jolla, California, USA, June 1995. ACM.
28. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *TCS*, 96(1):73–155, 1992.
29. M. Odersky and M. Zenger. Independently Extensible Solutions to the Expression Problem. In *FOOL*, January 2005.
30. B. C. d. S. Oliveira. Modular Visitor Components. In 23rd *ECOOP*, volume 5653 of *LNCS*, pages 269–293. Springer, 2009.
31. B. C. d. S. Oliveira and W. R. Cook. Extensibility for the Masses – Practical Extensibility with Object Algebras. In 26th *ECOOP*, volume 7313 of *LNCS*, pages 2–27. Springer, 2012.
32. B. C. d. S. Oliveira, A. Moors, and M. Odersky. Type Classes as Objects and Implicits. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, 25th *OOPSLA*, pages 341–360. ACM, October 2010.
33. B. C. d. S. Oliveira, S.-C. Mu, and S.-H. You. Modular Reifiable Matching: A List-of-Functors Approach to Two-Level Types. In B. Lippmeier, editor, 8th Haskell, pages 82–93. ACM, September 2015.
34. B. C. d. S. Oliveira, T. van der Storm, A. Loh, and W. R. Cook. Feature-Oriented Programming with Object Algebras. In Giuseppe Castagna, editor, 27th *ECOOP*, volume 7920 of *LNCS*, pages 27–51, Montpellier, France, 2013. Springer.
35. J. Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Comp. Surv.*, 27(2):196–255, 1995.

36. P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open Multi-Methods for C++. In C. Consel and J. L. Lawall, editors, 6th *GPCE*, pages 123–134. ACM, October 2007.
37. P Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Design and Evaluation of C++ Open Multi-Methods. *SCP*, 75(7):638 – 667, 2010.
38. R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 7th edition, 2009.
39. D. Rebernak, M. Mernik, P. R. Henriques, D. Carneiro da Cruz, and M. J. V. Pereira. Specifying Languages Using Aspect-oriented Approach: AspectLISA. *CIT*, 14(4):343–350, 2006.
40. T. Rendel, J. I. Brachthäuser, and K. Ostermann. From Object Algebras to Attribute Grammars. In A. P. Black and T. D. Millstein, editors, 28th *OOPSLA*, pages 377–395. ACM, October 2014.
41. J. C. Reynolds. User-Defined Types and Procedural Data Structures as Complementary Approaches to Type Abstraction. In S. A. Schuman, editor, *New Direc. Algo. Lang.*, pages 157–168. INRIA, 1975.
42. T. Rompf and M. Odersky. Lightweight Modular Staging: a Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In 9th *GPCE*, pages 127–136, Eindhoven, Holland, 2010. ACM.
43. T. Sheard and E. Pasalic. Two-Level Types and Parameterized Modules. *JFP*, 14(5):547–587, 2004.
44. Y. Solodkyy, G. Dos Reis, and B. Stroustrup. Open Pattern Matching for C++. In J. Järvi and C. Kästner, editors, *GPCE'13*, pages 33–42. ACM, October 2013.
45. I. Sommerville. *Software Engineering.* Addison-Wesley, 9th edition, 2011.
46. W. Swierstra. Data Types à la Carte. *JFP*, 18(4):423–436, 2008.
47. S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe Composition of Product Lines. In C. Consel and J. L. Lawall, editors, 6th *GPCE*, pages 95–104, Salzburg, Austria, 2007. ACM.
48. T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comp. Surv.*, 47(1):6:1–6:45, June 2014.
49. M. Torgersen. The Expression Problem Revisited. In M. Odersky, editor, 18th *ECOOP*, volume 3086 of *LNCS*, pages 123–143, Oslo (Norway), June 2004.
50. D. Vandevoorde, N. M. Josuttis, and D. Gregor. *C++ Templates: The Complete Guide.* Addison Wesley, 2nd edition, 2017.
51. P. Wadler. The Expression Problem. Java Genericity Mailing List, November 1998.
52. Y. Wang and B. C. d. S. Oliveira. The Expression Problem, Trivially! In 15th *Modularity*, pages 37–41, New York, NY, USA, 2016. ACM.
53. D. Wonnacott. Using Accessory Functions to Generalize Dynamic Dispatch in Single-Dispatch Object-Oriented Languages. In R. Raj and Y.-M. Wang, editors, 6th *COOTS*, pages 93–102. USENIX, February 2001.
54. M. Zenger and M. Odersky. Extensible Algebraic Datatypes with Defaults. In 6th *ICFP*, pages 241–252, Florence, Italy, 2001. ACM.
55. H. Zhang, Z. Chu, B. C. d. S. Oliveira, and T. van der Storm. Scrap Your Boilerplate with Object Algebras. In J. Aldrich and P. Eugster, editors, 29th *OOPSLA*, pages 127–146. ACM, October 2015.
56. H. Zhang, H. Li, and B. C. d. S. Oliveira. Type-Safe Modular Parsing. In B. Combemale, M. Mernik, and B. Rumpe, editors, 10th *SLE*, pages 2–13. ACM, October 2017.